Chapter 10 Abstract Classes and Interfaces

- These notes are a subset, for use in cosc1076 Programming 2, of the chapter 10 lecture notes from the prescribed textbook: Introduction to Java Programming by Y. Daniel Liang, Prentice-Hall, 2007

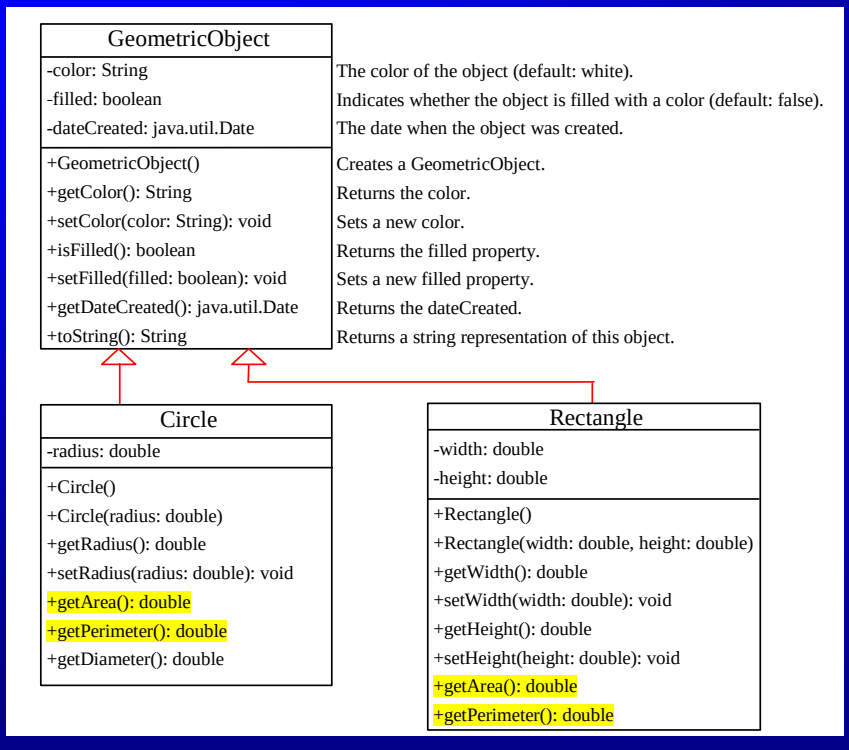
Objectives

- To design and use abstract classes (§10.2).
- To declare interfaces to model weak inheritance relationships (§10.4).
- To know the similarities and differences between an abstract class and an interface (§10.4.2).
- To declare custom interfaces (§10.4.3).
- To use wrapper classes (Byte, Short, Integer, Long, Float, Double, Character, and Boolean) to wrap primitive data values into objects (§10.5).
- To simplify programming using JDK 1.5 automatic conversion between primitive types and wrapper class types (§10.6).

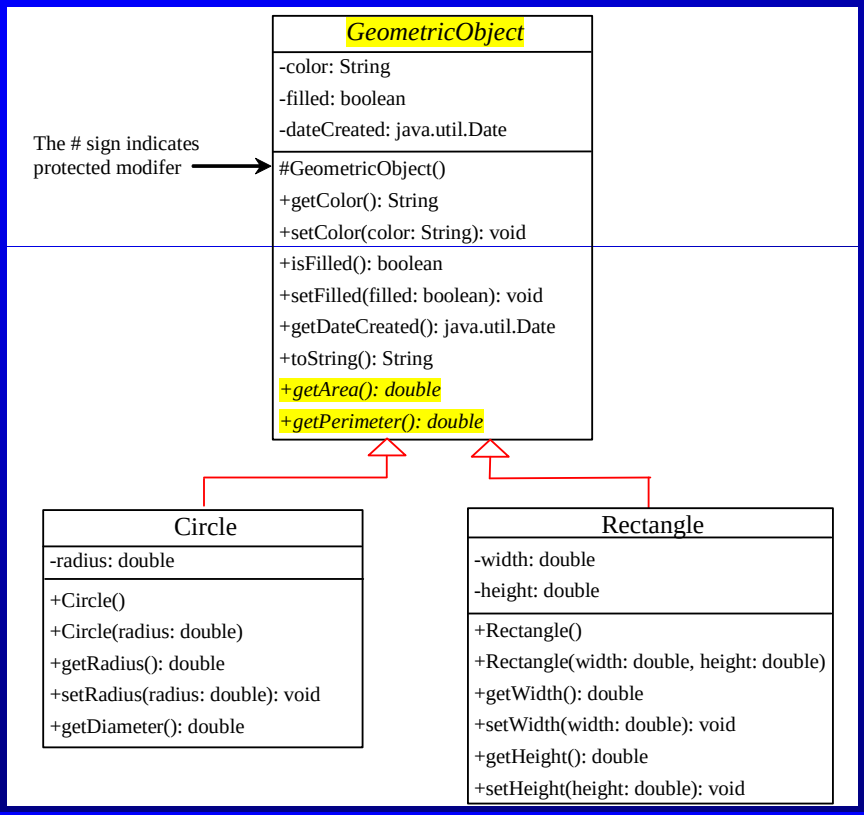
The `abstract` Modifier

- The `abstract` class
  - Cannot be instantiated
  - Should be extended and implemented in subclasses
- The `abstract` method
  - Method signature without implementation

From Chapter 9



Abstract Classes



GeometricObject

Circle

Rectangle

NOTE

An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be declared abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.

NOTE

An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

NOTE

A class that contains abstract methods must be abstract. However, it is possible to declare an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.

NOTE

A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

NOTE

A subclass can override a method from its superclass to declare it abstract. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be declared abstract.

NOTE

You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

```
GeometricObject[] geo = new
GeometricObject[10];
```

Example: Using the GeometricObject Class

- Objective: This example creates two geometric objects: a circle, and a rectangle, invokes the equalArea method to check if the two objects have equal area, and invokes the displayGeometricObject method to display the objects.

TestGeometricObject

Run

Interfaces

An *interface* is a classlike construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but an abstract class can contain variables and concrete methods as well as constants and abstract methods.

To distinguish an interface from a class, Java uses the following syntax to declare an interface:

```
public interface InterfaceName {
  constant declarations;
  method signatures;
}
```


Interface is a Special Class

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a variable, as the result of casting, and so on.

Example Using Interfaces

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

```
class Animal {
}

class Chicken extends Animal
    implements Edible {
  public String howToEat() {
    return "Fry it";
  }
}

class Tiger extends Animal {
}
```

```
class abstract Fruit
    implements Edible {
}
```

```
class Apple extends Fruit {
  public String howToEat() {
    return "Make apple cider";
  }
}

class Orange extends Fruit {
  public String howToEat() {
    return "Make orange juice";
  }
}
```


Implements Multiple Interfaces

```
class Chicken extends Animal implements Edible, Comparable {
  int weight;
  public Chicken(int weight) {
    this.weight = weight;
  }
  public String howToEat() {
    return "Fry it";
  }
  public int compareTo(Object o) {
    return weight – ((Chicken)o).weight;
  }
}
```

Creating Custom Interfaces, cont.

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

```
public class TestEdible {
  public static void main(String[] args) {
    Object[] objects = {new Tiger(), new Chicken(), new Apple()};
    for (int i = 0; i < objects.length; i++)
      showObject(objects[i]);
  }

  public static void showObject(Object object) {
    if (object instanceof Edible)
      System.out.println(((Edible)object).howToEat());
  }
}
```


Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be public static final	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Interfaces vs. Abstract Classes, cont.

All data fields are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

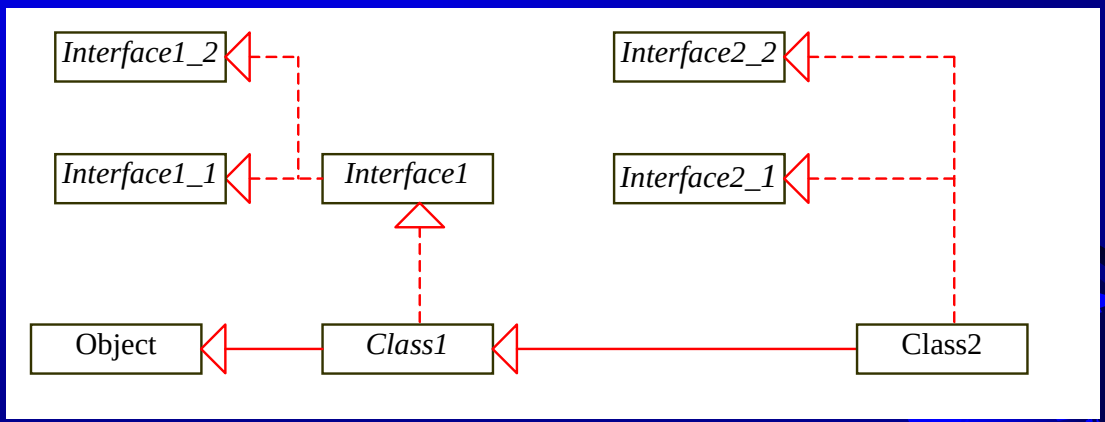
```
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

Interfaces vs. Abstract Classes, cont.

All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If an interface extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.



Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Caution: conflict interfaces

In rare occasions, a class may implement two interfaces with conflict information (e.g., two same constants with different values or two methods with same signature but different return type). This type of errors will be detected by the compiler.

Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features. How do you decide whether to use an interface or a class? In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person. So their relationship should be modeled using class inheritance. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface. See Chapter 10, "Object-Oriented Modeling," for more discussions.

General Purpose Interfaces

Suppose you want to design a generic method to find the larger of two objects. The objects can be students, dates, or circles. Since compare methods are different for different types of objects, you need to define a generic compare method to determine the order of the two objects. Then you can tailor the method to compare students, dates, or circles. For example, you can use student ID as the key for comparing students, radius as the key for comparing circles, and volume as the key for comparing dates. You can use an interface to define a generic compareTo method, as follows:

Example of an API Interface

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable {
  public int compareTo(Object o);
}
```

String and Date Classes

Many classes (e.g., String and Date) in the Java library implement Comparable to define a natural order for the objects. If you examine the source code of these classes, you will see the keyword implements used in the classes, as shown below:

```
public class String extends Object
    implements Comparable {
  // class body omitted
}
```

```
public class Date extends Object
    implements Comparable {
  // class body omitted
}
```

```
new String() instanceof String
new String() instanceof Comparable
new java.util.Date() instanceof java.util.Date
new java.util.Date() instanceof Comparable
```


Generic `max` Method

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Comparable max
      (Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(a)

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Object max
      (Object o1, Object o2) {
    if ((Comparable)o1).compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(b)

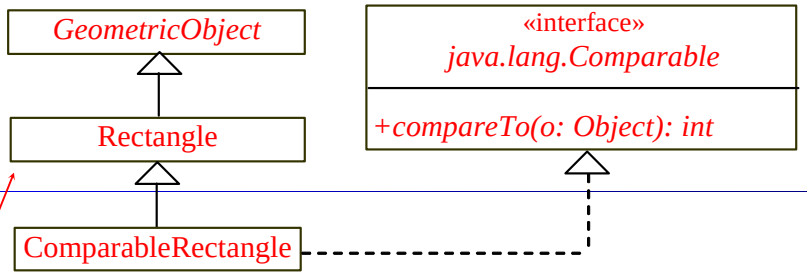
```
String s1 = "abcdef";
String s2 = "abcdee";
String s3 = (String)Max.max(s1, s2);
```

```
Date d1 = new Date();
Date d2 = new Date();
Date d3 = (Date)Max.max(d1, d2);
```

The return value from the max method is of the Comparable type. So, you need to cast it to String or Date explicitly.

Declaring Classes to Implement Comparable

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.



ComparableRectangle

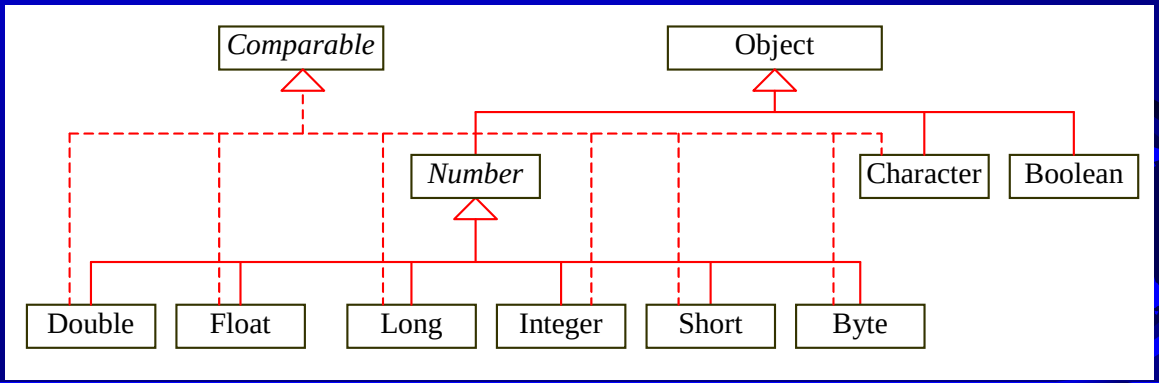
You cannot use the max method to find the larger of two instances of Rectangle, because Rectangle does not implement Comparable. However, you can declare a new rectangle class that implements Comparable. The instances of this new class are comparable. Let this new class be named ComparableRectangle.

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```


Wrapper Classes

- Boolean
- Character
- Short
- Byte
- Integer
- Long
- Float
- Double

NOTE: (1) The wrapper classes do not have no-arg constructors. (2) The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.



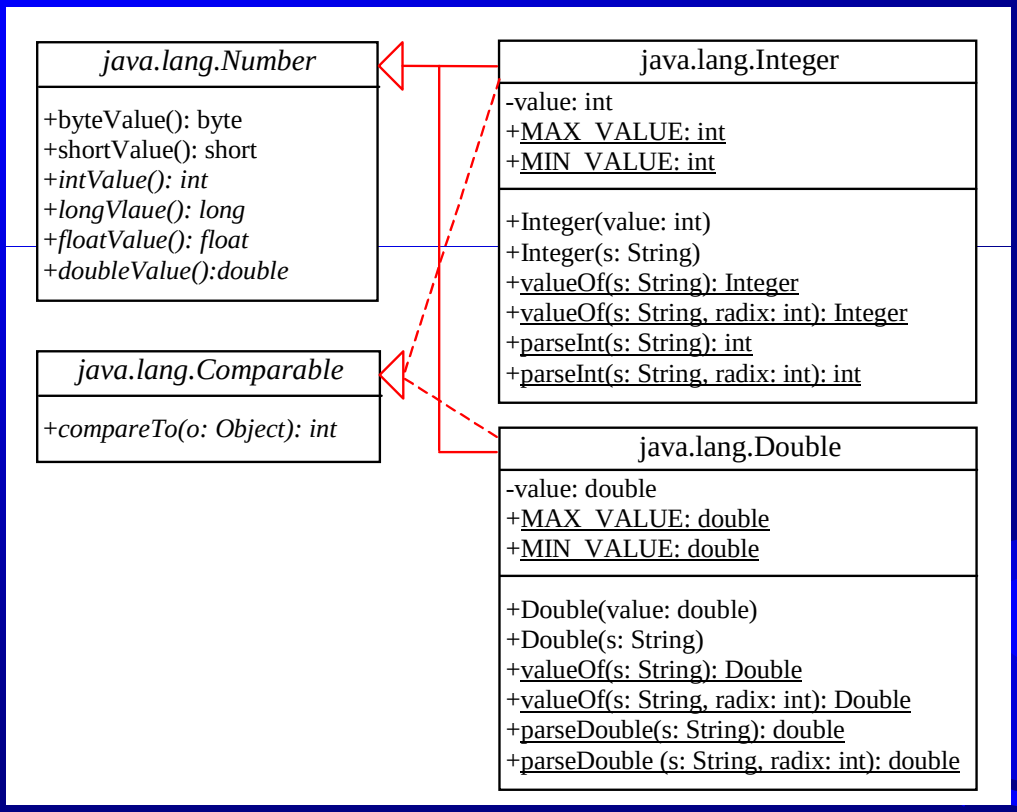
The toString, equals, and hashCode Methods

Each wrapper class overrides the toString, equals, and hashCode methods defined in the Object class. Since all the numeric wrapper classes and the Character class implement the Comparable interface, the compareTo method is implemented in these classes.

The Number Class

Each numeric wrapper class extends the abstract Number class, which contains the methods doubleValue, floatValue, intValue, longValue, shortValue, and byteValue. These methods "convert" objects into primitive type values. The methods doubleValue, floatValue, intValue, longValue are abstract. The methods byteValue and shortValue are not abstract, which simply return (byte)intValue() and (short)intValue(), respectively.

The `Integer` and `Double` Classes



The `Integer` Class and the `Double` Class

- Constructors
- Class Constants `MAX_VALUE`, `MIN_VALUE`
- Conversion Methods

Numeric Wrapper Class Constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value. The constructors for Integer and Double are:

public Integer(int value)

public Integer(String s)

public Double(double value)

public Double(String s)

Numeric Wrapper Class Constants

Each numerical wrapper class has the constants MAX_VALUE and MIN_VALUE. MAX_VALUE represents the maximum value of the corresponding primitive data type. For Byte, Short, Integer, and Long, MIN_VALUE represents the minimum byte, short, int, and long values. For Float and Double, MIN_VALUE represents the minimum *positive* float and double values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45), and the maximum double floating-point number (1.79769313486231570e+308d).

Conversion Methods

Each numeric wrapper class implements the abstract methods doubleValue, floatValue, intValue, longValue, and shortValue, which are defined in the Number class. These methods "convert" objects into primitive type values.

The Static valueOf Methods

The numeric wrapper classes have a useful class method, valueOf(String s). This method creates a new object initialized to the value represented by the specified string. For example:

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

The Methods for Parsing Strings into Numbers

You have used the parseInt method in the Integer class to parse a numeric string into an int value and the parseDouble method in the Double class to parse a numeric string into a double value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

JDK 1.5 Feature

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):

```
Integer[] intArray = {new Integer(2),
  new Integer(4), new Integer(3)};
```

(a)

Equivalent

```
Integer[] intArray = {2, 4, 3};
```

(b)

New JDK 1.5 boxing

```
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Unboxing